ICSC Qualification Round Submission

Harshil Tanna

June 2025

1 Problem A - Neural Network Components

We can identify the following within the context of restaurant satisfaction as:

- $\boldsymbol{w^{(1)}_{(21)}}$: A weight in the first hidden layer's weight matrix, this is denoted by the superscript (1). The subscript (21) suggests the weight links input node 2 with hidden layer node 1.
- $\boldsymbol{\Sigma}$: The weighted summation of all input signals given to the hidden layer neurons
- $\boldsymbol{f}$: The activation function that scales how strong the combined inputs and bias are before passing on to the next layer
- **Red Circle**: The input neurons: duration stayed and tip received
- **Orange Circle**: The fixed bias neuron for the first hidden neuron
- **Green Circle**: The output neuron quantifying customer satisfaction
- $\boxed{\textbf{Box A}}$: Hidden Layer
- $\boxed{\textbf{Box B}}$: Output Layer
- $\boldsymbol{\hat{y}}$: The final predicted customer satisfaction

2 Problem B - Cake Calculator

We are given a simple recipe for making cakes, where each cake requires:

- 100 units of flour
- 50 units of sugar

Given a supply of flour and sugar, we are to determine two things: the maximum number of cakes that can be made, and the amount of each ingredient left over after baking as many cakes as possible.

To simplify the problem, observe that **both** ingredients are necessary to bake a cake. Therefore, the maximum number of cakes is limited by whichever ingredient runs out first. This is known as the limiting ingredient.

Let F represent the available units of flour, and S the available units of sugar. The number of cakes that can be made using the available flour is $\left\lfloor \frac{F}{100} \right\rfloor$, and the number that can be made using the available sugar is $\left\lfloor \frac{S}{50} \right\rfloor$. Since both ingredients are required, the actual number of cakes that can be made is the minimum of these two values:

$$C = \min\left(\left\lfloor \frac{F}{100} \right\rfloor, \left\lfloor \frac{S}{50} \right\rfloor\right)$$

After baking C cakes, the amount of flour and sugar that remains unused can be computed as follows:

$$\text{Remaining flour} = F - 100 \times C$$

$$\text{Remaining sugar} = S - 50 \times C$$

This simple logic ensures that we maximise the number of cakes without exceeding the available inventory of either ingredient.

My Code:

```python
# Problem B - Cake Calculator
def cake_calculator(flour: int, sugar: int) -> list:
    """
    Calculates the maximum number of cakes that can be made and the
     leftover ingredients.

```

```
    Args:
        flour: An integer larger than 0 specifying the amount of available flour.
        sugar: An integer larger than 0 specifying the amount of available sugar.

    Returns:
        A list of three integers:
        [0] the number of cakes that can be made
        [1] the amount of leftover flour
        [2] the amount of leftover sugar

    Raises:
        ValueError: If inputs flour or sugar are not positive.
    """
    if flour <= 0 or sugar <= 0: # Handle non-positive integers
        raise ValueError("Flour and sugar must be positive integers")

    cakes = min(flour//100, sugar//50) # We use the limiting ingredient as our guide for number of cakes that can be baked
    flour_left = flour - (cakes*100) # Subtract any used flour
    sugar_left = sugar - (cakes*50) # Subtract any used sugar

    return [cakes, flour_left, sugar_left] # Final values returned as an array
```

3 Problem C - The School Messaging App

Question 1: Why Use Variable-Length Codes?

One approach would be to assign the same number of bits to every character. However, not all characters are used equally. If we assign **shorter codes to more frequent characters**, and **longer codes to less frequent ones**, we can reduce the total number of bits used overall.

Example: Suppose we send messages that mostly use the letter `A`, but occasionally use `Z`. In a fixed length code, both might be encoded with 3 bits: `A = 000`, `Z = 111`.

But since `A` is used more often, we could assign it a shorter code, such as `A = 0`. Conversely, we could give the rarer `Z` a longer code, such as `Z = 111`.

Now, if we want to send the message `AAA`, fixed length encoding would use $3 \times 3 = 9$ bits. But with variable-length encoding, it would only need 3 bits: `000`, thus saving 6 bits just on that small message.

Over many messages, this adds up and lets us send more content using less data.

Question 2: Calculating Entropy

To compute the entropy H, we plug in the probabilities:

$$\begin{aligned}
H = &-(0.20\log_2 0.20 + 0.15\log_2 0.15 + 0.12\log_2 0.12 + 0.10\log_2 0.10 \\
&+ 0.08\log_2 0.08 + 0.06\log_2 0.06 + 0.05\log_2 0.05 + 0.05\log_2 0.05 \\
&+ 0.04\log_2 0.04 + 0.03\log_2 0.03 + 0.02\log_2 0.02 + 0.10\log_2 0.10) \\
\approx\ & 3.324 \text{ bits}
\end{aligned}$$

This entropy value represents the **theoretical lower bound** on the average number of bits required per character for optimal encoding.

Question 3: Comparing Average Code Length to Entropy

To evaluate efficiency, we calculate the average code length L using:

$$L = \sum_{i=1}^{n} p_i \times \text{length}(c_i)$$

$$\begin{aligned}
L &= 0.20(3) + 0.15(3) + 0.12(3) + 0.10(4) + 0.08(4) + 0.06(4) \\
&\quad + 0.05(3) + 0.05(4) + 0.04(4) + 0.03(4) + 0.02(4) + 0.10(4) \\
&= 3.48 \text{ bits}
\end{aligned}$$

This is very close to the theoretical entropy of 3.324 bits, showing that Fano encoding is highly efficient and a good choice for this message dataset.

4 Problem D - Word Search Puzzle

The task was to generate a 10×10 word search puzzle that hides a list of given words in a grid of uppercase letters. The words must appear in a continuous sequence.

My Approach

I opted to allow diagonal, vertical, horizontal and reversed placement of words to enhance complexity. However, this comes with a significant roadblock that not all word placements in the grid will be valid. Each word needs to fit entirely in a straight line, and the challenge is reduced to checking all valid placements while ensuring words do not conflict with each other.

Step 1: Direction Vectors To cover all possible placements, I defined eight directions using pairs of (dx, dy) values. These represent the direction a word can be extended in the grid:

$$(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)$$

This includes diagonals, rows, and columns in both forward and reverse.

Step 2: Precomputing Valid Positions Next, I wrote a helper function that, for each word, calculates all valid starting positions and directions where the word would stay entirely within bounds. This gave me a list of possible placements for each word, which I could later test one by one.

Step 3: Backtracking Algorithm To place all the words in the grid without overlap, I used backtracking. The idea was to place the first word in one of its valid positions, then try to place the second word, and so on. If a word could not be placed without conflict, I backtrack by removing the previous word and trying a different option.

This guarantees that all words are placed only if a valid arrangement exists.

Step 4: Handling Conflicts During placement, if a cell is already occupied by a letter that doesn't match the current word's character at that position, I consider it a conflict and undo the entire placement for that word. This way, no word overwrites another.

Step 5: Random Filler Letters Once all the words were successfully placed, I filled the remaining empty cells with random uppercase letters using `random.choice`.

My Code:

```python
# Problem D - Word Search Puzzle
import random
import string

def create_crossword(words):
    """
    Generate a 10x10 word search puzzle containing the given words. Places words in all directions, alongside their reversed variants.

    Args:
        words: A list of words to include in the puzzle.

    Returns:
        A 2D array (list of lists) representing the word search puzzle.
    """
    # Problem D - Word Search Puzzle
    SIZE = 10  # Grid size: 10x10
    words = [w for w in words if w]  # Remove blanks

    # Validate words
    if not words:
        raise ValueError("No words provided.")

    for w in words:
        if not w.isalpha():
            raise ValueError(f"Invalid word '{w}': only letters allowed")
        if len(w) > SIZE:
            raise ValueError(f"Word too long for {SIZE}x{SIZE} grid: '{w}' ({len(w)})")

    # Uppercase after validation
    words = [w.upper() for w in words]
    # Start with longer words
    words.sort(key=len, reverse=True)

    # Define all 8 directions (horizontal, vertical, diagonal for both forward and reverse directions)
    directions = [(dx, dy) for dx in [-1, 0, 1] for dy in [-1, 0, 1] if not (dx == dy == 0)]

    # Find all valid placements of a word within the grid
    def get_valid_placements(word):
        placements = []
        for dx, dy in directions:
            for x in range(SIZE):
                for y in range(SIZE):
```

```
                    end_x, end_y = x + dx * (len(word) - 1), y + dy * (len(word) - 1)
                    if 0 <= end_x < SIZE and 0 <= end_y < SIZE:
                        placements.append((x, y, dx, dy))
        return placements

    # Store valid placements for each word
    word_placements = {w: get_valid_placements(w) for w in words}
    if any(len(p) == 0 for p in word_placements.values()):
        raise ValueError("Could not fit words in grid")  # Exit early if a word cannot fit in any direction

    # Create empty grid
    grid = [['' for _ in range(SIZE)] for _ in range(SIZE)]

    # Try to place a word in the grid at a given position and direction
    def place_word(word, x, y, dx, dy):
        written = []  # Only cells we set from empty to a letter
        for i, ch in enumerate(word):
            cx, cy = x + dx * i, y + dy * i
            cell = grid[cx][cy]
            if cell not in ('', ch):  # Conflict
                for (vx, vy) in written:  # Undo partial character placement
                    grid[vx][vy] = ''
                return None
            if cell == '':
                grid[cx][cy] = ch
                written.append((cx, cy))
        return written

    # Undo helper
    def undo_word(written):
        for (vx, vy) in written:
            grid[vx][vy] = ''

    # Backtracking algorithm to place all words without conflict
    def backtrack(index):
        if index == len(words):
            return True  # All words have been placed
        word = words[index]
        random.shuffle(word_placements[word])  # Randomise placement options
        for x, y, dx, dy in word_placements[word]:
            written = place_word(word, x, y, dx, dy)
            if written is not None:
                if backtrack(index + 1):
                    return True
                undo_word(written)  # Undo and try next
        return False

    if not backtrack(0):
        raise ValueError("Could not fit words in grid")  # Could not solve

    # Fill remaining empty spaces with random letters
```

```
    for i in range(SIZE):
        for j in range(SIZE):
            if grid[i][j] == '':
                grid[i][j] = random.choice(string.ascii_uppercase)

    return grid
```

5 Problem E - Functional Completeness of NAND

The **NAND** gate ("**NOT AND**") outputs 0 only when both inputs are 1. Its truth table is:

a	b	NAND(a, b)
0	0	1
0	1	1
1	0	1
1	1	0

To prove that **NAND** is functionally complete, we need to show that we can express the basic Boolean operations **NOT**, **AND**, and **OR** using only the **NAND** operation.

1. Constructing NOT using NAND

To get **NOT** from **NAND**, we feed the same input into both inputs of the gate:

$$\text{NOT}(a) = \text{NAND}(a, a)$$

`Explanation:` Since **NAND** outputs 0 only when both inputs are 1, this gives 1 when $a = 0$, and 0 when $a = 1$. In other words, it negates the input.

2. Constructing AND using NAND

Recall that **NAND** is the negation of **AND**. So, to get **AND**, we apply **NOT** to the result of **NAND**:

Using our previous definition of **NOT**...

$$\text{AND}(a, b) = \text{NOT}(\text{NAND}(a, b)) = \text{NAND}(\text{NAND}(a, b), \text{NAND}(a, b))$$

`Explanation:` This double-**NAND** negates the **NAND** output, yielding the true **AND**.

3. Constructing OR using NAND

To construct the **OR** operation using only **NAND** gates, we start by applying **De Morgan's Law**, which states:

$$a \vee b = \neg(\neg a \wedge \neg b)$$

This identity is useful because it rewrites **OR** in terms of **AND** and **NOT** — both of which we've already expressed using **NAND**.

We can now translate this into **NAND**-based gates step-by-step:

1. First, compute $\neg a$ and $\neg b$ using:

$$\neg a = \text{NAND}(a, a) \quad \text{and} \quad \neg b = \text{NAND}(b, b)$$

2. Then compute the **AND** of $\neg a$ and $\neg b$:

$$\neg a \wedge \neg b = \text{NAND}(\text{NAND}(a, a), \text{NAND}(b, b)) \text{ then negated}$$

3. Remember that the equation above is representative of an **AND** gate, therefore, applying **NOT** reverts it back to a **NAND** gate. Hence, we can simply apply **NAND** to $\text{NAND}(a, a)$ and $\text{NAND}(b, b)$

$$a \vee b = \neg(\neg a \wedge \neg b) = \text{NAND}(\text{NAND}(a, a), \text{NAND}(b, b))$$

`Explanation:` Each $\text{NAND}(x, x)$ produces $\neg x$. Then the outer NAND gives the negation of their AND, which by De Morgan's law is exactly $a \vee b$. Therefore, the OR function is successfully built using only NAND gates.

Conclusion

We have successfully constructed the three fundamental Boolean operations – **NOT**, **AND** and **OR** – using only the **NAND** gate. Since any Boolean expression can be formed using combinations of these three, we conclude that:

$$\boxed{\textbf{NAND} \text{ is functionally complete.}}$$